Theory Comput. Systems **40**, 327–353 (2007)
DOI: 10.1007/s00224-006-1311-1

Theory of Computing Systems

Sorting Stably, in Place, with $O(n \log n)$ Comparisons and $O(n)$ Moves

Gianni Franceschini

Dipartimento di Informatica, Università di Pisa,
Largo B, Pontecorvo 3, 56127 Pisa, Italy
francesc@di.unipi.it

Abstract. We settle a long-standing open question, namely whether it is possible to sort a sequence of n elements stably (i.e., preserving the original relative order of the equal elements), using $O(1)$ auxiliary space and performing $O(n \log n)$ comparisons and $O(n)$ data moves. Munro and Raman stated this problem in *J. Algorithms* (**13**, 1992) and gave an in-place but unstable sorting algorithm that performs $O(n)$ data moves and $O(n^{1+\varepsilon})$ comparisons. Subsequently (*Algorithmica*, **16**, 1996) they presented a stable algorithm with these same bounds. Recently, Franceschini and Geffert (FOCS 2003) presented an *unstable* sorting algorithm that matches the asymptotic lower bounds on all computational resources.

1. Introduction

In the *comparison model* the only operations allowed on the totally ordered domain of the input elements are the comparison of two elements and the transfer of an element from one cell of memory to another. Therefore, in this model it is natural to measure the efficiency of an algorithm with three metrics: the number of comparisons it requires, the number of element moves it performs and the number of auxiliary memory cells it uses, besides the ones strictly necessary for the input elements. It is well known that in order to sort a sequence of n elements, at least $n \log n - n \log e$ comparisons have to be performed in the worst case. Munro and Raman [15] set the lower bound for the number of moves to $\lfloor 3/2n \rfloor$. An *in-place* or *space-optimal* algorithm uses $O(1)$ auxiliary memory cells. In the general case of input sequences with repeated elements, an important requirement for a sorting algorithm is to be *stable*: the relative order of equal elements in the final sorted sequence is the same found in the original one.

1.1. *Previous Work*

The sorting problem is fundamental in computer science and has been widely studied from the very beginning. The Heapsort [19] is the first space-optimal sorting algorithm performing $O(n \log n)$ comparisons in the worst case. However, this algorithm is unstable and the number of element moves performed in the worst case is $O(n \log n)$. The existence of a sorting algorithm that is stable and comparison- and space-optimal was proven by Pardo [16] with the introduction of the first linear-time, stable, in-place merging algorithm.

If we consider the partition-based approach, the original *Quicksort* [5] needs a recursion stack. However, the stack can be eliminated (see [3], [1] and [18]) and space optimality can be achieved. A *stable and comparison- and space-optimal* sorting can be derived using in-place stable selection and partition algorithms like the ones in [7] and [8].

Katajainen and Pasanen [9] presented an unstable sorting requiring $o(n \log n)$ moves in the worst case while guaranteeing in-placeness and $O(n \log n)$ comparisons. That algorithm performs $O(n \log n / \log \log n)$ data moves in the worst case.

Concerning the sorting algorithms with an optimal number of data moves, the classical selection sort operates in place and performs $O(n)$ moves in the worst case but it is not stable and performs $O(n^2)$ comparisons in the worst case. An improvement in the number of comparisons came from Munro and Raman [13] with a generalization of the Heapsort performing $O(n^{1+\varepsilon})$ comparisons in the worst case. Finally, a stable algorithm with these same bounds was presented in [14].

If the space optimality is given up, the address-table sort [10] performs an optimal number of comparisons and moves but it requires $O(n)$ auxiliary cells of memory. However, it can be easily modified to achieve the stability, and the space requirement has been reduced to $O(n^{\varepsilon})$ by a variant of samplesort [13].

Recently, Franceschini and Geffert [4] presented an *unstable* sorting algorithm that matches the asymptotic lower bounds on all the computational resources, space, comparisons and data moves.

1.2. *Our Result*

In this paper we settle a long-standing open question explicitly stated by Munro and Raman in [13], namely whether it is possible to sort a sequence of n elements stably, using $O(1)$ auxiliary space, performing $O(n \log n)$ comparisons and $O(n)$ data moves. So far, the best-known algorithm for stable, in-place sorting with $O(n)$ moves was the one presented by Munro and Raman in [14], performing $O(n^{1+\varepsilon})$ comparisons in the worst case.

2. The Algorithm in Brief

Two basic techniques are very common when space efficiency of algorithms and data structures in the comparison model is the objective. The first is *bit stealing* [12]: a bit of information is encoded in the relative order of a pair of distinct input elements.

The second technique is *internal buffering* [11], in which some of the input elements are used as placeholders in order to simulate a working area and permute the other elements at less cost. The internal buffering is one of the most powerful techniques for space-efficient algorithms and data structures. However, it is easy to understand how disruptive the internal buffering is when the stability of the algorithm is an objective. If the placeholders are not distinct, the original order of identical placeholders can be lost using the simulated working area. As a witness of the clash between stability and internal buffering technique, we can cite the difference in complexity between the first in-place, linear-time merging algorithm, due to Kronrod [11], and the first stable one by Pardo [16].

2.1. *What's New*?

The crucial difference with other space- and comparison-optimal but unstable sorting algorithms performing a near-optimal or optimal number of moves, like [9] or [4], is in how the internal buffering technique is used. In those algorithms a large internal buffer with $\Theta(n)$ elements has to be found. In order to have such a large buffer, the internal buffering process is iterated $O(\log n)$ times, halving the size of the sub-problem, until it becomes so small that it can be treated without internal buffering.

The problem in this process is that it ignores completely the characteristics of the input sequence, namely, the number of distinct elements. If the sequence has a very limited number of distinct elements, the conventional approach for internal buffering cannot do anything good. On the other hand, it is probable that this extreme characteristic of the sequence can be exploited in some unconventional bottom-up way.

Our algorithm follows an approach that can be well synthesized as "adaptive." Using some sophisticated new techniques, we introduce a sorting method that *adapts to the number d of distinct elements of the input sequence*. In particular, for what concerns the harder case where the sequence has only a small number of distinct elements, that method allows us to sort a sequence using two kinds of internal buffers:

- An internal buffer with only $\Theta(d)$ distinct elements is needed to sort the whole sequence stably and within our resource bounds. That requires an efficient algorithm to extract a set of $\Theta(d)$ distinct elements from the input sequence.
- An internal buffer with $\Theta(n)$, not necessarily distinct, elements. The original order of this large buffer can be *completely recovered* after it has been used. As we will see, this normally unachievable task (as in [9] and [4]) is a direct consequence of the small number of distinct elements in the sequence.

Our strategy for the development of a stable sorting algorithm matching the asymptotic lower bounds on all the computational resources can be synthesized in four major points.

2.2. *Stealing Bits*

In Section 3 we show how to extract from the input sequence $\Theta(n/\log n)$ pairs of distinct elements stably and within our computational bounds. They will be used for encoding purposes with the basic bit-stealing technique. We make use of the stable, in-place selection and the stable, in-place partitioning algorithms described in [7] and [8]. We

obtain a "slow" auxiliary encoding memory that we will use in the rest of the paper to sort the remaining $m = O(n)$ elements laid down in sequence A. Finally, after A is sorted, the $\Theta(n/\log n)$ elements devoted to information encoding will be sorted using the normal in-place, stable mergesort.

2.3. *Extracting Distinct Elements*

In Section 4 we show how to extract as many distinct elements as we can from the $m = \Theta(n)$ elements in the sequence A left to be sorted at the end of Section 3. We start gathering the elements with rank less than or equal to $s = \Theta(n/\log^3 n)$ from the other ones. Let the resulting sequence be $A'A''$. This can be done stably and within our target bounds using once again the stable, in-place selection and partitioning algorithms proposed in [7] and [8]. Let d be the number of distinct elements among the ones in A''.

Then, we show how to extract $b = \Theta(\min(d, n/\log^3 n))$ distinct elements from A'', stably and within our computational bounds, by exploiting the elements in A'. Basically, we grow a data structure (encoding its auxiliary data in the auxiliary encoding memory built in the previous point) at the left end of the space, where all the elements of A' initially reside. The structure is built while we scan A'' from right to left for distinct elements (their distinctness being evaluated using the structure built so far). The structure is basically a semi-dynamic dictionary optimally searchable but with a slower insertion time. When an element of A'' qualifies for the structure, it is exchanged with the first (leftmost) available element in A' (that does not belong to the structure) and the structure is updated. At the end of the scan we have to extract the elements in A'' that have been used as placeholders (as in the internal-buffering technique) when a new element was inserted in the structure. For the sake of stability, we have to extract those scattered elements maintaining the original order that equal occurrences had before the construction of the structure. In the end we will have b distinct elements in a subsequence B at the right end of the space (the set of distinct element grows in left end of the space and it is finally moved when it is complete). We will be left with the problem of sorting the subsequence C with the remaining elements from A'' by exploiting the buffer of distinct elements B (and the set of encoded bits stolen in the Section 3). After C is sorted, the $b = \Theta(\min(d, n/\log^3 n))$ buffer elements in B can be sorted using the normal in-place mergesort. (We do not need the stable one, since B contains distinct elements.)

2.4. *Sorting in Presence of Many Distinct Elements*

In Section 5 we show how to proceed when the subsequence A'' has "many" distinct elements, meaning $\Omega(n/\log^3 n)$ distinct elements.

We divide our sorting problem into $\Theta(\log^3 n)$ sub-problems of size $\Theta(n/\log^3 n)$ and show how to solve those sub-problems assuming the availability of a sufficient number of distinct elements to be used as placeholders, that is, in case $b = \Theta(n/\log^3 n)$. Those buffer elements are the ones in B obtained in Section 4.

First, we introduce a structure that can sort $O(b)$ elements in $O(b \log n)$ comparisons and $O(b)$ moves *stably* by exploiting B and the auxiliary encoded memory obtained in Sections 4 and 3, respectively. As in the buffer-extraction procedure of Section 4, this structure may be seen as a semi-dynamic dictionary but in that case we have almost completely opposite targets. When we extract distinct elements, we want a structure

totally compact with efficient search but with possibly slow insertion. When we sort C in the presence of many distinct elements, we want a structure that is not compact, as it exploits the set of distinct buffer elements, but that has an efficient insertion, in particular for what concerns the complexity bound on the number of moves (we can only afford $O(1)$ moves in an amortized sense for each insertion).

Finally, after we used the elements in B and the structure to build $\Theta(\log^3 n)$ runs of sorted elements out of the original sequence C left to be sorted from Section 4, we have to merge these runs stably and within our computational bounds. To this purpose, we introduce a multi-way stable merging technique requiring a very limited number of placeholders to deliver the final sorted sequence.

2.5. *Sorting in Presence of Few Distinct Elements*

In Section 6 we show how to deal with the hardest case, namely when, after the extraction of the buffer elements in Section 4, we are left to sort a sequence C with "few" distinct elements, meaning a sequence with $b = o(n/\log^3 n)$ distinct elements. First, we partition the sequence into three zones $C'YC''$ around a pivot element. (The occurrences equal to the pivot will be in zone Y.) That is done because we are going to need an efficient way to distinguish between the sequence of active elements V (that will be impersonated first by C' and then by C'') and two types of buffer elements, a collection (in B) of few but distinct buffer elements and another collection of many but not necessarily distinct elements (that will be impersonated first by C'' and then by C').

Then we show how to exploit the scarcity of distinct elements in the sequence V grouping the identical elements lying in sub-sequences of size $\Theta(b \log^2 n)$ and how to acquire and encode in the auxiliary encoding memory (Section 3) a linked structure traversing the groups of clustered equal elements in V in sorted order.

Finally, we show how to use the groups and the encoded linked structure to permute first C' using YC'' as the working zone and then C'' using $C'Y$ *without disrupting the* (*sorted*) *order of* C'.

3. Stealing Bits

As we mentioned in the Introduction, with the *bit-stealing* technique (see [12]) the value of a bit is encoded in the relative order of two distinct elements (e.g., the increasing order for 0 and the decreasing order for 1). In this section we show how to collect $\Theta(n/\log n)$ pairs of distinct elements, stably and within our computational bounds.

The *rank* of an element x_i in a *sequence* $S = x_1 \cdots x_t$ is the cardinality of the multiset

$$\{x_j \in S \mid x_j < x_i \text{ or } (x_j = x_i \text{ and } j \leq i)\}.$$

The *rank* of an element x in a *set* $\mathscr{S}$ is similarly defined. Let $r = \lceil n/\log n \rceil$ and let π' and π'' be, respectively, the element with rank r and the element with rank $n - r + 1$ in the input sequence. We want to *stably* and in-place partition the input sequence into five zones $J'P'AP''J''$ such that, for each $j' \in J'$, $p' \in P'$, $a \in A$, $p'' \in P''$ and $j'' \in J''$,

we have that

$$j' < p' = \pi' < a < p'' = \pi'' < j''.$$

That can be done in $O(n)$ comparisons and $O(n)$ moves using the stable, in-place selection and the stable, in-place partitioning of Katajainen and Pasanen [7], [8].

Zones J' and J'' can be sorted stably and in place in $O(n)$ time simply using a stable in-place mergesort (e.g., [17]). If there are no elements in A, we are done since the input sequence is already sorted. Otherwise we are left with the unsorted subsequence A and with a set $\mathscr{M}$ of $r = \Theta(n/\log n)$ pairs of distinct elements, that is,

$$\mathscr{M} = \left\{(Q'[1], Q''[1]), (Q'[2], Q''[2]), \ldots, (Q'[r], Q''[r])\right\},$$

where $Q' = J'P'$ and $Q'' = P''J''$.

The starting addresses of Q' and Q'' can be maintained in two locations of auxiliary memory (we can use $O(1)$ auxiliary locations) and so, for any i, we can retrieve the addresses of the elements of the ith pair in $O(1)$ operations. Therefore, we can view $\mathscr{M}$ as a *collection of encoding words* of t bits each, for any t. Those encoding words can be used pretty much as if they were normal ones. We have to pay attention to the costs of using encoding bits or encoding words, though: *reading* an encoding word of t bits takes t comparisons, *changing* it costs t comparisons and $O(t)$ data moves in the worst case or $O(1)$ moves amortized if we perform a sufficiently long sequence of increments by one (see [2], the binary counter analysis). It is worth noting that we could have chosen the ranks of π' and π'' as cr and $n - cr + 1$ for any constant c, so that the number of encoded bits would be cr without changing the asymptotic complexity bounds of the algorithm.

Therefore, if m is the size of A, we can make the following assumption:

Assumption 1. *We can use an* auxiliary encoding memory $\mathbb{M}$ *consisting of* $\Theta(r/\log m)$ *words of* $\lceil \log m \rceil$ encoding bits *each and with the following cost model. For any word* w, *for any* $q \leq \lceil \log m \rceil$ *and for any group* g *of* q *bits of* w:

- *retrieving the value encoded in* g *requires* q *comparisons in the worst case*;
- *changing the value encoded in* g *requires* $\Theta(q)$ *moves in the worst case*.

Hence, if we are able to solve the following problem over the sequence A, we are able to solve the original problem.

Problem 1. Under Assumption 1, sort the sequence A of m elements stably, using $O(1)$ locations of auxiliary memory, performing $O(m \log m)$ comparisons and $O(m)$ data moves.

In the following sections we use the auxiliary encoding memory $\mathbb{M}$ as normal auxiliary memory for numeric values. We will declare explicitly any new auxiliary data (indices, pointers...) stored in $\mathbb{M}$.

4. Extracting a Set of Distinct Elements

In this section we show how to go from sequence A to $J'''P'''CB$ such that:

Property 1.

(i) *For any elements $j \in J'''$, $p \in P'''$ and $q \in CB$, we have that $j < p < q$.*
(ii) *J''' is in sorted order.*
(iii) *The element with rank $\lceil r/\log^2 m \rceil + 1$ in A is in P''' together with all the other elements equal to it.*
(iv) *Let d be the number of distinct elements in CB, where B contains*

$$b = \min(d, \lceil |CB|/\log^3 m \rceil)$$

distinct elements.
(v) *Any two equal elements in $J'''P'''CB$ are in the same relative order as in A.*

After we show how to obtain the new sequence $J'''P'''CB$ satisfying Property 1 within our target bounds, we will be left with the problem of sorting CB. The elements in B will be used in Sections 5 and 6 as in the technique of *internal buffering* [11]. Basically, some of the elements are used as placeholders to simulate a working area in which the other elements can be permuted efficiently. If the placeholders are not distinct, stability becomes a difficult task since the original order of identical placeholders can be lost using the simulated working area. The elements in B are distinct so we do not have to worry, as long as we can sort $O(|C|)$ elements with $o(|C|)$ placeholders (Section 5). However, as we will see in Section 6, if $|B|$ is too small, we have to also use a larger internal buffer whose entire original order, not only the relative order of equal elements, has to be preserved.

4.1. *Main Cycle of the Buffer Extraction*

We first present the main cycle of the algorithm for the creation of $J'''P'''CB$. The main cycle depends on a complex structure that we will introduce later in Section 4.2.

Before we start, let us recall the basic technique for space-efficient block exchange. From a block $X = x_1 \cdots x_t$ of t consecutive elements we can obtain the reverse $X^R = x_t \cdots x_1$ in linear time and in place simply exchanging x_1 with x_t, x_2 with x_{t-1} and so forth. Two consecutive blocks X and Y, possibly of different sizes, can be exchanged in place and in linear time with three block reversals, since $YX = (X^R Y^R)^R$.

The main cycle of the buffer extraction procedure has three phases.

4.1.1. *First Phase*: *Collecting some Placeholders*. In the first phase we extract some elements, possibly non-distinct, that will help in the process of collecting the set of distinct elements that will reside in B.

First, we select the element of rank $\lceil r/\log^2 m \rceil + 1$ in A. Then we partition A according to that element.

We obtain a new sequence $A'P'''A''$ that clearly satisfies points (i) and (iii) in Property 1, with $J''' = A'$ and $CB = A''$. The selection and the partitioning can be done in place and stably using once again the linear-time algorithms proposed by Katajainen

and Pasanen [7], [8]. Therefore, point (v) in Property 1 is also satisfied. If A'' is void, we sort A' using the in-place, stable mergesort and we are done. Otherwise, we leave A' as it is and we proceed with the second phase.

4.1.2. *Second Phase*: *Collecting the Distinct Elements*. Throughout this phase we continue to denote by A the evolving sequence of m elements. We have that $A = A'P'''A''$ right after the first phase. Let us denote with h the index of the rightmost location of P'''.

We maintain two indices i and i' initially set, respectively, to 1 and m. The following steps are repeated until $i > \lceil |A''| / \log^3 m \rceil$ or $i' = h$:

1. SEARCH($A[i']$, $A[1 \cdots i-1]$).
2. If $A[i']$ is not in $A[1 \cdots i-1]$, exchange $A[i']$ and $A[i]$, PROCESS($A[1 \cdots i]$) and increase i by one.
3. Decrease by one i'.

At the end of this second phase, we have collected $b = \min(d, \lceil |A''|/\log^3 m \rceil)$ distinct elements in $A[1 \cdots b]$ (d is the number of distinct elements in A''). How the procedures SEARCH and PROCESS work will be explained in Section 4.2.

4.1.3. *Third Phase*: *Collecting the Placeholders Back*. After the second phase, the first b elements residing in $A'P'''$ at the end of the first phase are scattered in the subsequence $A[h+1 \cdots m]$. Therefore, point (v) in Property 1 is no longer satisfied by the current sequence A. We have to collect them back.

First, we partition the subsequence $A[h+1 \cdots m]$ according to $A[h]$. Let CA''' be the resulting sequence, where for any $a \in A'''$ and $c \in C$, we have that $a \leq A[h] < c$. We once again use the linear time, stable partitioning algorithm from [8].

Then we reverse A''', recovering the original order holding before the second phase, we sort it using the stable in-place mergesort and we exchange it with $A[1 \cdots b]$.

After that, the resulting sequence A respects all the points in Property 1.

Lemma 1. *Under Assumption* 1, *the buffer extraction algorithm operates in place*, *Property* 1 *holds for the resulting sequence*, *and the comparisons and moves performed are*, *respectively*, $O(mX_s + X_p + m)$ *and* $O(Y_p + m)$, *where*

- X_s *upper bounds the number of comparisons of each invocation of* SEARCH *in step* 1,
- X_p *and* Y_p *are*, *respectively*, *the total number of comparisons and moves performed by the b invocations of* PROCESS *in step* 2.

Proof. First phase. We apply the stable, in-place, linear-time selection and partitioning algorithm proposed in [7] and [8]. If we already have to sort the elements in A' (because nothing has to be done for A''), we can use the normal stable, in-place mergesort since $|A'| = O(m/\log^3 m)$.

Second phase. The cycle is iterated $O(m)$ times, hence the total cost of the invocations of SEARCH is $O(mX_s)$ comparisons. During the cycle Step 2 can be executed $O(r/\log^2 m)$ times and, excluding the costs of PROCESS, its complexity is $O(1)$. There-

fore the total cost of Step 2 is $o(r) = o(m)$ comparisons and moves. Step 3 contributes another $O(m)$ arithmetic operation.

Third phase. It consists simply in one application of the partitioning algorithm in [8], a constant number of applications of block reversing and exchanging and the final application of the stable, in-place mergesort to the first $b = O(m/\log^3 m)$ elements in A. □

4.2. *Managing a Growing Set of Distinct Elements Compactly*

In this section we describe the structure we use to perform efficiently the operations SEARCH and PROCESS in Steps 1 and 2 of the second phase of the buffer extraction algorithm.

The structure has two levels:

- the *routing level*, which directs the searches, and
- the *collection* level, which contains the majority of the elements in the structure.

First we give the solution to an abstract problem. Then we describe the structure and, in particular, how to reduce the managing of the routing level to an instance of the abstract problem. The abstract problem (and its solution) will come in handy again in Section 5 where we describe another two-level structure but with different target bounds and characteristics than the one in this section.

4.2.1. *Abstract Problem.*

We want to solve the following abstract problem.

Problem 2. We are given two disjoint sets: $\mathscr{R}$ with *routing elements* and $\mathscr{F}$ with *filler elements*. The following hypotheses hold:

(i) Routing and filler elements belong to the same totally ordered, possibly infinite universe.
(ii) At any time we are presented with a new routing element to be included in $\mathscr{R}$.
(iii) At most $\rho < m$ elements will be included in $\mathscr{R}$.
(iv) At the beginning $|\mathscr{R}| = 1$, the unique routing element is in the first location followed by the fillers.
(v) At any time $|\mathscr{F}|/|\mathscr{R}| > \log \rho$. The possible growth of $\mathscr{F}$ is not a concern, as new filler elements will eventually be added after the current last element.
(vi) We can use an auxiliary memory $\hat{\mathbb{M}}$ of $\Theta(\rho)$ words of $\log \rho$ bits each to store auxiliary data.

The task is to manage the growth of $\mathscr{R}$ so that the following properties hold:

(a) At any time $\mathscr{R}$ and $\mathscr{F}$ are stored in a zone $\mathscr{Z}$ of $|\mathscr{R}| + |\mathscr{F}|$ contiguous memory locations.
(b) At any time $\mathscr{R}$ can be searched with $O(\log|\mathscr{R}|)$ comparisons and a constant number of accesses to $\hat{\mathbb{M}}$.
(c) When $\mathscr{R}$ is complete, the total number of comparisons, moves and accesses to $\hat{\mathbb{M}}$ performed is $O(|\mathscr{R}| \log^2 \rho)$.

How to solve Problem 2. We maintain in $\mathcal{Z}$ a sequence $\mathcal{S}$ of contiguous *segments* containing $\sigma = O(\log \rho)$ elements each. Let $\mathcal{S} = S_1 S_2 \cdots S_{t-1} S_t$ be the sequence of segments at a given time.

A segment can contain both routing and filler elements. A segment with i routing elements has the following structure:

$$a_1 a_2 \cdots a_{i-1} a_i f_1 f_2 \cdots f_{\sigma - i},$$

where any a_j is a routing element and any f_j is a filler element. There is at least one routing element for each segment.

For any segment we have to be able to discern between its routing and filler elements. Standing solely on the hypotheses of Problem 2, we must assume that it is not possible to classify any element in $\mathscr{R} \cup \mathscr{F}$ as routing or filler simply by inspection. Therefore any segment S_i is associated with an integer counting the number of routing elements in it. The buffer separator technique used in [4] cannot be used for this purpose because, as we will see, in at least one instance of the abstract Problem 2 there is no relation between the filler elements and the routing elements of a segment.

As we will see shortly, the number of segments will always be a power of 2. The routing elements are maintained in sorted order throughout the whole $\mathcal{S}$, that is, the routing elements of every segment are in sorted order and, for any two routing elements a', a'' such that $a' \in S_i$, $a'' \in S_j$ and $i < j$, we have that $a' \leq a''$.

A routing element can be easily searched in $O(\log|\mathscr{R}|)$ comparisons and a constant number of accesses to $\hat{\mathbb{M}}$: first, do a binary search over the first elements of the segments (that are all routing elements); then read in $\hat{\mathbb{M}}$ the number of routing elements of the only segment selected with the previous step and search in it. Therefore, property (b) of Problem 2 holds.

While inserting new routing elements, the invariants on $\mathcal{S}$ can be maintained with a variation of the well-known density-based algorithm in [6]. The nodes of an implicit binary tree are associated with subsequences of $\mathcal{S}$. The root of the tree is associated with the whole sequence $\mathcal{S}$. The left child of the root is associated with $S_1 S_2 \cdots S_{t/2-1} S_{t/2}$, the right child with $S_{t/2+1} S_{t/2+2} \cdots S_{t-1} S_t$ and so forth (the number of segments will always be a power of 2). Therefore, there is a leaf for each segment. A node v has two attributes: the level $l(v)$ (the leaves are at level 0, the root is at level $\log t$) and the number $d(v)$ of routing elements contained in the subsequence associated with v. Each level has a threshold: level i has threshold $\tau_i = \sigma - i$. The tree has $2t - 1 \leq 2|\mathscr{R}| \leq 2\rho$ nodes and therefore all the attributes of the nodes of the tree can be stored in $\hat{\mathbb{M}}$. (Actually, they can also be calculated at rebalancing time, without storing or encoding anything, but we do not need to do this.)

When a new routing element a is inserted in the proper segment $S' = a_1 a_2 \cdots a_{i-1} a_i g_1 g_2 \cdots g_{\sigma - i}$ containing $i < \sigma$ routing elements, the first filler element g_1 is moved after the current last element of $\mathcal{Z}$ and the process ends with the segment $S' = a_1 a_2 \cdots a_j a a_{j+1} a_{i-1} a_i g_2 \cdots g_{\sigma - i}$ for some j.

Otherwise, if S' is full, we find the lowest ancestor v of the leaf associated with S' such that $d(v) \leq 2^{l(v)} \cdot \tau_{l(v)}$ and redistribute the routing elements *evenly* in the subsequence associated with v. That costs $O(2^{l(v)} \cdot \tau_{l(v)})$. If not even the root of the tree satisfies the condition, then the number of segments is doubled, a new implicit tree with a larger root is used and the redistribution can be performed. By hypothesis (v) in Problem 2 we

know that, for any $|\mathscr{R}|$, there are enough filler elements to create the new t segments. Therefore, property (a) of Problem 2 holds.

Finally, with a simple analysis, similar to the one in [6], it can be proved that property (c) of Problem 2 holds. After the redistribution of the elements in the subsequence associated with v, for each descendant u of v we have that $d(u) \leq 2^{l(v)} \cdot \tau_{l(v)}$. In particular that holds for the children of v. Before the rebalancing, there was a child u' of v such that $d(u') > 2^{l(u')} \cdot \tau_{l(u')}$. Therefore, before v needs to be rebalanced again there will have to be at least

$$2^{l(u'')}(\tau_{l(u'')} - \tau_{l(v)})$$

insertions in the subsequence associated with any child u'' of v. Since the rebalancing of v cost $O(2^{l(v)} \cdot \tau_{l(v)})$, we have that the amortized cost *relatively to level* $l(v)$ of the insertion that triggered the rebalancing is

$$O\left(\frac{2^{l(v)} \cdot \tau_{l(v)}}{2^{l(u')}(\tau_{l(u')} - \tau_{l(v)})}\right) = O(\tau_{l(v)}).$$

Since there are $\log t$ levels, the complete amortized cost is $O(\sigma \log t) = O(\log \rho \log t) = O(\log^2 \rho)$. Therefore we can conclude that Problem 2 is solved.

4.2.2. *The Structure.* In order to manage the growth of the set of buffer elements in the second phase of the buffer extraction algorithm, we have to solve the following problem.

Problem 3. Under Assumption 1, we have to handle the growth of a set $\mathscr{B}$ of at most $\lceil r/\log^2 m \rceil = O(m/\log^3 m)$ distinct elements so that the following properties hold:

(a) At any time, $\mathscr{B}$ is stored in $|\mathscr{B}|$ contiguous memory locations.
(b) At any time, $\mathscr{B}$ can be searched with $O(\log m)$ comparisons.
(c) When $\mathscr{B}$ is complete, the total number of comparisons and moves performed is $O(|\mathscr{B}| \log^3 m)$.

Let us give our solution for Problem 3. Let B be the (growing) zone of the memory in which $\mathscr{B}$ will be maintained. The auxiliary encoding memory $\mathbb{M}$ has $\Theta(r/\log m)$ words of $\lceil \log m \rceil$ bits each. Since $|\mathscr{B}| \leq \lceil r/\log^2 m \rceil$, it is easy to associate with every $x \in \mathscr{B}$ a constant number h of words of auxiliary data. We can allocate in $\mathbb{M}$ an array I_B of $\lceil r/\log^2 m \rceil$ entries of h words each and maintain the following invariant:

The element in position ith *in* B *has its auxiliary data stored in the* ith *entry of* I_B. (1)

For the sake of description, we skip that kind of detail in the algorithm and implicitly assume that an element is always moved together with its encoded $O(1)$ words of auxiliary data. (We consider the unusual cost model for $\mathbb{M}$ in the analysis.)

At any time, B is divided into two contiguous zones R and H. The elements of the routing level are in R (together with some elements of the collection level that will act as fillers, as we will see).

Buckets. Each routing element a is associated with a set of elements $\beta(a)$ in the collection level that we will call a *bucket*. Let a' and a'' be two consecutive (in the sorted order) routing elements: for each $x \in \beta(a')$ we have that $a' < x < a''$. For what concerns the number of elements in a bucket, we have that $4\lceil \log m \rceil \leq |\beta(a)| \leq 8\lceil \log m \rceil$.

Let us focus on a single bucket β and let $even_\beta$ and odd_β be the set of the elements of β with even and odd rank, respectively. The elements in odd_β are stored *in sorted order in a contiguous zone of memory in H* while the elements in $even_\beta$ are stored *in R and may be scattered*. They will play a role similar to the one of buffer elements but more powerful because they will be searchable at any moment of the lifetime of the structure.

Pointers are used to keep track of the elements in $even_\beta$: each $o' \in odd_\beta$ has a pointer to its successor $succ(o') \in even_\beta$ and $succ(o')$ has a pointer to o'. (Obviously, if $|\beta|$ is odd, the successor of the largest element in odd_β does not belong to the set $even_\beta$.)

The routing element a associated with β has a pointer to the location of odd_β and odd_β has a pointer to a. Assuming that we are able to maintain the layout we just introduced for the buckets, we can show a way to satisfy property (b) of Problem 3.

Lemma 2. *A bucket can be searched with $O(\log m)$ comparisons.*

Proof. Searching for an element u in a bucket β is straightforward. First we search in odd_β. If $u \notin odd_\beta$, let $o \in odd_\beta$ be the predecessor of u in odd_β. We access $succ(o)$ using the pointer of o. If $succ(o)$ is not equal to u, the search ends. In total we have to access $O(1)$ words of auxiliary information in $\mathbb{M}$. From Assumption 1, the thesis follows. □

Sub-zones. H has to accommodate the set odd_β for any bucket β. By the upper and lower bounds for the number of elements in a bucket, we know that $2\lceil \log m \rceil \leq \left|odd_\beta\right| \leq 4\lceil \log m \rceil$ for any odd_β.

All the odd_β of size i are maintained in a contiguous sub-zone $H_{i-2\lceil \log m \rceil+1}$ of H (the sub-zone H_j contains sets of size $j+2\lceil \log m \rceil-1$). Therefore, there are $z = 2\lceil \log m \rceil+1$ sub-zones. We have that $H = H_1 H_2 \cdots H_{z-1} H_z$, that is, the sub-zones are in increasing order by the size of sets they contain.

For any H_i, the first set odd_β may be *rotated*, that is, it may have its first $i + 2\lceil \log m \rceil - 1 - j$ elements at the right end of H_i and the last j at the left end, for any j called *index of rotation* of H_i.

Some sub-zones may be void. For any zone, we store in $\mathbb{M}$ its starting address in H and its index of rotation. If the indices of rotation are known, the particular case of a routing element a having a bucket β with the set odd_β in the first position of its sub-zone can be treated simply (e.g., with an extra flag for each routing element to recognize the particular case).

Basic operations. We are going to need two basic operations on the sub-zones: SLIDE_BY_ONE(i) and MOVE_BACK(o).

- With SLIDE_BY_ONE(i) all the sub-zones H_j with $j \geq i$ are rotated by one position to the right (assuming that there is a free location at the right end of H). The execution of this operation is obvious.

- With MOVE_BACK(o) the set o of size $4\lceil \log m \rceil$ (the maximum size possible) is moved from H_z to H_1.
  1. o is exchanged with the second set in H_z.
  2. o is exchanged with the portion of the first set in H_z residing at the left end (H_z can have rotation index > 0).
  3. For each H_i, $2 \leq i \leq z-1$, o is exchanged with the first $4\lceil \log m \rceil$ elements of H_i.
  4. o is exchanged with the portion (if any) of the first set of H_1 residing at the right end (H_1 can also have rotation index > 0).

Lemma 3. *The operations* SLIDE_BY_ONE(i) *and* MOVE_BACK(o) *can be executed with* $O(\log^3 m)$ *moves and comparisons.*

Proof. In SLIDE_BY_ONE(i) we have to access and modify $\Theta(z-i)$ pointers in $\mathbb{M}$ storing the rotation indices of the sub-zones involved. That requires $\Theta((z-i)\log m) = O(\log^2 m)$ moves and comparisons in the worst case. Moreover, we have to update the pointers of $O(z-i)$ buckets whose *odd* sets are stored in the $\Theta(z-i)$ sub-zones and were rotated before the execution of this operation. Therefore, SLIDE_BY_ONE(i) requires $O(\log^3 m)$ moves and comparisons in the worst case.

Since the minimum size of a set in H ($2\lceil \log m \rceil$) is of the same order of the maximum size ($4\lceil \log m \rceil$), in MOVE_BACK(o) we have to access and modify $O(z)$ pointers stored in $\mathbb{M}$. Analogously to the previous case, we have to update the pointers of $\Theta(\log m)$ buckets whose *odd* sets are moved in this operation. Therefore, MOVE_BACK(o) requires $O(\log^3 m)$ moves and comparisons in the worst case. □

Maintaining the invariants for the collection level. Let us show how the invariants on B introduced so far can be maintained when an element u has to be inserted in a bucket β associated with a routing element a placed somewhere in zone R.

Let us suppose $|\beta| = p < 8\lceil \log m \rceil$, and let i be the rank of u in $\beta \cup \{u\}$. There are two phases: in the first phase we reorganize the space to make room for the new element; in the second phase we rearrange the elements of β, since the arrival of u may change odd_β and $even_\beta$ substantially.

- *Space reorganization.* If p is odd then $even_\beta$ increases by one and odd_β remains of the same size. We invoke SLIDE_BY_ONE(1) to free a location between R and H and put u in that location temporarily.

  Otherwise, if p is even then odd_β increases by one and $even_\beta$ remains of the same size.
  1. We invoke SLIDE_BY_ONE($p/2 - 2\lceil \log m \rceil + 2$) in order to have a free location between $H_{p/2-2\lceil \log m \rceil+1}$ (the sub-zone that contained odd_β before the insertion of u in β) and $H_{p/2-2\lceil \log m \rceil+2}$ (the new sub-zone of odd_β after the insertion) and we put u in the free location temporarily.
  2. We exchange odd_β with the last set in $H_{p/2-2\lceil \log m \rceil+1}$.
  3. We exchange odd_β with the portion (if any) of the first set of $H_{p/2-2\lceil \log m \rceil+1}$ residing at the right end of the sub-zone.
  4. After odd_β is joined with u, we exchange $odd_\beta \cup \{u\}$ with the portion of the first set in $H_{p/2-2\lceil \log m \rceil+2}$ residing at the left end of the new sub-zone.

- *Rearrangement*. Let $\beta' = \beta \cup \{u\}$, $\Delta_o = \{o \in odd_\beta \mid u < o\}$ and $\Delta_e = \{e \in even_\beta \mid u < e\}$. If the rank of u in β' is odd, we have that

  $$odd_{\beta'} = \{u\} \cup (odd_\beta - \Delta_o) \cup \Delta_e \quad \text{and} \quad even_{\beta'} = (even_\beta - \Delta_e) \cup \Delta_o.$$

  Similarly, if the rank of u in β' is even, we have that

  $$odd_{\beta'} = (odd_\beta - \Delta_o) \cup \Delta_e \quad \text{and} \quad even_{\beta'} = \{u\} \cup (even_\beta - \Delta_e) \cup \Delta_o.$$

  Given the definition of $odd_{\beta'}$ and $even_{\beta'}$, the rearrangement is a simple sequence of exchanges between an element in H and an element in R. For each exchange we have to access one pointer.

Let us suppose β is full (before the insertion of u). We have to split $\beta \cup \{u\}$ into two buckets β', β'' and the median element a'. β' and β'' will contain, respectively, the $4\lceil \log m \rceil$ smallest and the $4\lceil \log m \rceil$ largest elements of $\beta \cup \{u\}$. a' will be a new routing element and its bucket will be β''.

- *Space reorganization*. First, we make room for a new routing element invoking SLIDE_BY_ONE(1) and put u in the free location between R and H. Second, we invoke MOVE_BACK(odd_β) to move odd_β in H_1.
- *Rearrangement*. We have to reorganize the disposition of the elements in $\beta \cup \{u\}$ for the splitting. However, we already know that the starting address of $odd_{\beta'}$ will be the same of odd_β after MOVE_BACK(odd_β) and the starting address of $odd_{\beta''}$ will be $2\lceil \log m \rceil$ locations further. If the rank of u in $\beta \cup \{u\}$ is $4\lceil \log m \rceil + 1$ then u is the new routing element to be inserted in R and we do not have to do any reorganization.

  Otherwise, we exchange u with the element of rank $4\lceil \log m \rceil + 1$ and then we proceed to rearrange the other elements in the same way we do when β is not full. In any case, by the end of the rearrangement process, we have the new routing element a' in the location between R and H, ready to be inserted in the routing level.

When we have to insert an element in a bucket that is full, we split the bucket and we are left with a new routing element to be inserted in the routing level.

Lemma 4. *Maintaining the invariants for the collection level costs $O(\log^3 m)$ moves and comparisons in the worst case.*

Proof. When $|\beta| = p < 8\lceil \log m \rceil$, the cost of SLIDE_BY_ONE(i) dominates the complexity of the space reorganization. For what concerns the rearrangement, we have to access $O(\log m)$ pointers in $\mathbb{M}$. Therefore, by Lemma 3, when we have to insert an element in a bucket that is not full, we pay $O(\log^3 m)$ moves and comparisons in the worst case.

When β is full, the cost of the execution of MOVE_BACK(odd_β) dominates the complexity of the space reorganization in this case. For the rearrangement, we have to access $O(\log m)$ pointers in $\mathbb{M}$. Therefore, by Lemma 3, when we have to insert an element in a bucket that is full, we pay $O(\log^3 m)$ moves and comparisons to split the bucket. □

The routing level. With the organization of RH presented so far, we are able to satisfy all the hypotheses in Problem 2. As expected, $\mathscr{R}$ contains the routing elements produced by splitting the buckets and $\mathscr{F}$ contains the elements in $\bigcup_\beta even_\beta$.

Hypotheses (i)–(iv) are obviously satisfied. For each routing element there is a bucket β with at least $4\lceil \log m \rceil$ elements and at least $2\lceil \log m \rceil$ of those elements belong to $\mathscr{F}$. Hence, Hypothesis (v) is satisfied. Concerning Hypothesis (vi), we can use the encoded memory $\mathbb{M}$.

Therefore, the solution to the abstract Problem 2 can be used and we are able to manage the growth of the set of routing elements so that the following properties hold:

- At any time, all the routing elements and the ones in $\bigcup_\beta even_\beta$ are maintained compactly in zone R.
- At any time, the routing level can be searched with $O(\log m)$ comparisons and a constant number of accesses to $\mathbb{M}$ (and hence $O(\log m)$ comparisons in total).
- In the routing level there is a slowdown factor $O(\log m)$ because we use the auxiliary encoded memory. However, we know that

  $$|\mathscr{R}| < \rho = O\left(\frac{|\mathscr{B}|}{\log m}\right).$$

  Therefore, when the routing level is complete, the total number of comparisons and moves performed building it is $O(|\mathscr{B}| \log^2 m) = O(m)$.

Joining those properties and Lemmas 2 and 4 we can conclude that Problem 3 is solved. Therefore, by Lemma 1, we have that:

Theorem 1. *Under Assumption* 1, *a sequence* $J'''P'''CB$ *satisfying Property* 1 *can be obtained from* A, *stably, using* $O(1)$ *locations of auxiliary memory, performing* $O(m \log m)$ *comparisons and* $O(m)$ *moves in the worst case.*

5. Sorting with Many Distinct Elements

In this section we show how to sort the subsequence CB of a sequence $J'''P'''CB$ satisfying Property 1 and with $b = |B| = \lceil |CB|/\log^3 m \rceil$. First, in Section 5.1, we show how to sort b elements stably, using $O(1)$ auxiliary space, with $O(b \log m)$ comparisons and $O(b)$ moves, under Assumption 1 and using another b distinct elements as placeholders (from B). Then, in Section 5.2, we show how to sort CB using the technique in Section 5.1 and a multi-way stable merging technique requiring a very limited number of placeholders.

5.1. *Sorting b Elements with b Distinct Placeholders*

For the sake of simplicity, let us suppose that $b = 5b'$, for an integer b'. Let D be the sequence of b elements to be sorted and let B be the sequence of b placeholders. D is divided into five contiguous subsequences of b' elements each, $D = D_1D_2D_3D_4D_5$. Each D_i is sorted using B as we will describe shortly, and the final sorted sequence is

obtained merging the subsequences in-place, stably and in linear time (e.g., using the merging algorithm described in [17]).

To sort each D_i, we use a structure with the same basic subdivision of the one in Section 4.2: a *routing level*, directing the searches, and a *collection level*, containing the majority of the elements. After all the elements in D_i are inserted, the structure is traversed to move the elements back in D_i stably and in sorted order.

5.1.1. *The Structure*. First we describe the logical organization of the collection level. Then we show how to embed the collection level into the internal buffer B and how to store in $\mathbb{M}$ its auxiliary data. Finally, we show how to maintain the invariants and how, even in this case, the routing level can be seen as a particular instance of the abstract Problem 2.

The collection level. Each routing element a is associated with a small balanced search tree $\mathcal{T}(a)$ in the collection level. Let a' and a'' be two consecutive (in the sorted order) routing elements: for each $x \in \mathcal{T}(a')$ we have that $a' \leq x \leq a''$.

Let us consider a generic tree $\mathcal{T}$. Concerning the number of elements in a leaf l of $\mathcal{T}$ we have that

$$\lceil \log m \rceil \leq |l| \leq 2\lceil \log m \rceil. \tag{2}$$

On the other hand, concerning the number of elements in an internal node $u \in \mathcal{T}$ we have that

$$\lceil \sqrt{\log m} \rceil \leq |u| \leq 2\lceil \sqrt{\log m} \rceil. \tag{3}$$

$\mathcal{T}$ has five levels and hence it contains at least $\log^3 m$ elements.

The elements in any leaf of $\mathcal{T}$ *are not in sorted order*; they are in *insertion order*: a new element of a leaf is inserted in the last position regardless for its rank among the other elements. There is no auxiliary encoded data associated with any single element of a leaf. Instead, a *bit mask* of $2\lceil \log m \rceil$ bits is associated with the whole leaf, one bit for each possible position, with the expected meaning: *the* ith *bit is equal to one if and only if there is an element in position* i.

The elements in any internal node v *are not in sorted order* either. However, a small encoded pointer of $O(\log \log m)$ bits is associated with any of them. Those pointers are used to maintain a small linked list in which the elements of v are maintained in sorted order. There is also a small encoded pointer to the head of the linked list and, of course, an encoded pointer of $O(\log m)$ bits to each child of the node.

Embedding and encoding. The internal buffer B is divided into four contiguous zones $RC'C''W$, such that

$$|R| = b' - (2\lceil \log m \rceil + 1),$$
$$|C'| = |C''| = 2b',$$
$$|W| = 2\lceil \log m \rceil + 1.$$

R, C' and C'' will be devoted to the embedding of routing elements, internal nodes and leaves of the trees in the collection level, respectively. W will be used as working area.

C'' is logically divided into *allocation units* of $2\lceil \log m \rceil$ positions (and elements) each. By the logical definition of the collection level, we know that for each leaf there is only a bit mask of $2\lceil \log m \rceil$ bits. Therefore we allocate in $\mathbb{M}$ an array $I_{C''}$ of $2b'/2\lceil \log m \rceil$ entries with $2\lceil \log m \rceil$ bits (two words) each.

Similarly, C' is logically divided into allocation units of $2\lceil \sqrt{\log m} \rceil$ positions (and elements) each. This time we allocate in $\mathbb{M}$ an array $I_{C'}$ of $2b'$ entries with three words each. It is a little more than we need but we can afford it (Assumption 1 and $b' < \lceil r/\log^2 m \rceil$).

Concerning W, we allocate in $\mathbb{M}$ an array I_W of $|W|$ entries with three words each.

Finally, we do not allocate any auxiliary encoded data for zone R right now, as it will be organized as a particular instance of abstract Problem 2.

As we will see, for both C' and C'', the allocation units will be occupied from left to right and never released. Therefore, two simple counters are the whole auxiliary information we need in order to manage the allocation units. Finally, it is worth noting that the sizes of R, C' and C'' are beyond the need. Obviously, other choices are possible but they can only lower the constant factors in the complexity bounds.

Maintaining the invariants. Let us suppose that an element x has to be inserted in a tree $\mathcal{T}$ in the collection level. x is routed toward a leaf l as expected: the linked list of the current internal node is scanned to find the rightmost element less than or equal to x and the process continues in the corresponding child. When l is reached, the bit-mask is scanned and the leftmost position occupied by a placeholder is found. (For that purpose a simple counter would do the job; as we will see, the bit mask will be essential when the leaf is split and when the structure is visited.) Finally, the corresponding bit is set to one and the placeholder and x are exchanged.

If l is full we have to split it in order to maintain invariant (2) on the number of elements. We allocate the first free allocation unit in C''. (The allocation process is a simple increment of an index, as we said.) That will contain the new leaf l'. Then we execute the following steps:

1. Find the rank $r(x)$ of x in the sequence lx (a simple scan of leaf l).
2. For $i = 2\lceil \log m \rceil + 1$ to 1:
   (a) If $i = r(x)$ then exchange x with $W[i]$ (a placeholder).
   (b) Otherwise, find the element y with maximum rank (among the ones still in l) scanning l and its bit mask, and set to zero the bit of y. Then, exchange y with $W[i]$ (again, a placeholder).
3. Exchange the first $\lceil \log m \rceil$ elements in l with $W[1 \cdots \lceil \log m \rceil]$, and set the bit mask of l to $1^{\lceil \log m \rceil}0^{\lceil \log m \rceil}$.
4. Exchange the first $\lceil \log m \rceil$ elements in l' with $W[|W| - \lceil \log m \rceil + 1 \cdots |W|]$, and set the bit mask of l' to $1^{\lceil \log m \rceil}0^{\lceil \log m \rceil}$.

After that, we have to insert the element x' of medium rank (that is still located in the $(\lceil \log m \rceil + 1)$th position of W) in the parent of l; let it be u. If u is not full, we simply follow the inner linked list of u until the rightmost (in the list order that is also the sorted order) element x'' less than or equal to x' is found. Then, we insert x' after x'' in the list and set its child pointer to the starting address of l'.

We suppose u is full, we have to split it in order to maintain invariant (3). We allocate a new unit u' in C', and we execute the following steps assuming that every time an element is moved from a location of C' to another in W, its auxiliary data (child pointer...) stored in the encoded array $I_{C'}$ is also moved to the corresponding position in I_W:

1. Follow the linked list of u until the rightmost element x'' less than or equal to x' is found. Let $r(x'')$ be the rank of x'' in the linked list. Exchange x' with $W[r(x'')+1]$.
2. Exchange the first $r(x'')$ elements in the inner list of u with the first $r(x'')$ elements of W (the first element in the list is exchanged with $W[1]$, the second one with $W[2]$ and so forth).
3. Exchange the last $2\lceil\sqrt{\log m}\rceil - r(x'')$ elements in the list of u with the last $2\lceil\sqrt{\log m}\rceil - r(x'')$ of W.
4. Exchange $W[1\cdots\lceil\sqrt{\log m}\rceil]$ with the first $\lceil\sqrt{\log m}\rceil$ elements of u and initialize its list.
5. Exchange $W[|W|-\lceil\sqrt{\log m}\rceil+1\cdots|W|]$ with the first $\lceil\sqrt{\log m}\rceil$ elements of u' and initialize its list.

After that, the element in position $\lceil\sqrt{\log m}\rceil+1$ of W is inserted in the parent of u and the process is iterated. If even the root of the tree has to be split, its medium rank element is inserted in R.

Lemma 5. *Under Assumption* 1, *the data structure can be built using* $O(1)$ *auxiliary space*, $O(b\log m)$ *comparisons and* $O(b)$ *moves*.

Proof. $\mathcal{T}$ has $O(1)$ levels and each internal node has $O(\sqrt{\log m})$ elements. Hence, the total number of comparisons we pay to scan the linked lists during the search for the position of x is $O(\sqrt{\log m}\log\log m)$. Scanning the bit mask of l costs $O(\log m)$ comparisons. Therefore, we pay $O(\log m)$ comparisons in order to find the position of x in $\mathcal{T}$.

If l is not full, the insertion of x in it costs only $O(1)$ moves, since we have to modify a bit of the bit mask and exchange x with the placeholder in l.

If l is full, let us analyze the steps of the procedure for splitting a leaf. Step 1 is a simple scan and it takes $O(\log m)$ comparisons. Step 2(a) is just a comparison of integer values. Step 2(b) takes a scan with $O(\log m)$ comparisons and $O(1)$ moves. Steps 2(a) and 2(b) are executed $O(\log m)$ times and then the total cost of step 2 is $O(\log^2 m)$ comparisons and $O(\log m)$ moves. Finally, steps 3 and 4 are simple scans and exchanges and take $O(\log m)$ comparisons and moves.

Then we have to insert x' in u, the parent of l. If u is not full we have to pay $O(\sqrt{\log m}\log\log m)$ comparisons to follow its inner list and $O(\log m)$ moves to update the pointers.

If u is full, let us analyze the steps of the procedure for splitting an internal node. We have to remember that every time an element of an internal node is moved its auxiliary encoded information in $I_{C'}$ is moved too.

Step 1 scans the inner list of u and exchange one element of u, that takes $O(\sqrt{\log m}\log\log m+\log m)$ comparisons and $O(\log m)$ moves. Steps 2 and 3 are

just a series of exchanges of the remaining elements in u and they take $O(\sqrt{\log m}\log m)$ moves and comparisons (remember, also the auxiliary encoded data in $I_{C'}$ is moved). Finally, steps 4 and 5 do exchanges of $O(\sqrt{\log m})$ elements formerly contained in u and initialize two inner lists. Hence they cost $O(\sqrt{\log m}\log m)$ moves and comparisons.

After the splitting of u, the process is iterated for its $O(1)$ ancestors. Given the above worst-case costs and the fact that each leaf and each internal node has, respectively, $O(\log m)$ and $O(\sqrt{\log m})$ elements, it is obvious to derive the amortized costs by inserting an element into $\mathcal{T}$. That is, there are $O(\log m)$ comparisons and $O(1)$ moves in the amortized sense.

If even the root of $\mathcal{T}$ has to be split, we insert a new routing element in R. To organize R we use the solution to Problem 2 in Section 4.2.1. All the hypotheses in Problem 2 are satisfied. Obviously, $\mathscr{R}$ contains the elements produced by splitting a tree in the collection level and $\mathscr{F}$ the placeholders initially in R. Hypotheses (i)–(iv) are easily satisfied. For each routing element, there is a tree $\mathcal{T}$ with at least $\log^3 m$ elements, therefore Hypothesis (v) is satisfied. We use the auxiliary encoded memory $\mathbb{M}$ to satisfy Hypothesis (vi).

Given the cost model in Assumption 1 and the solution to Problem 2 in Section 4.2.1, the thesis follows. □

5.1.2. *Traversing the Structure.* After the construction of the structure, D_i contains placeholders. Traversing the structure is pretty standard. We maintain five pointers p_r, p_1, p_2, p_3, p_4 in auxiliary memory; p_r points to the rightmost visited routing element in R and p_1, p_2, p_3, p_4 point to the internal nodes in the current visiting path of the tree of the routing element pointed by p_r. For each p_j, we have to maintain a small pointer s_j to the rightmost visited element in the internal linked list of the node pointed by p_j. Actually, any pointed element (by p_r or by any s_j) is immediately exchanged with the leftmost placeholder in D_i, and only its auxiliary encoded data is still accessible to guide the visit.

Each leaf l is visited in the following way:

1. Compute the number j of elements in l scanning its bit mask.
2. For $i = 1$ to j:
   (a) Find the element x with minimum rank (among the ones still in l) scanning l and its bit mask.
   (b) Set its bit in the bit mask of l to zero.
   (c) Exchange x with the leftmost placeholder in D_i.

By the cost model in Assumption 1, it is immediate to prove that the traversing phase ends with all the elements back in D_i in stable sorted order and that the whole traversing phase takes $O(1)$ auxiliary locations, $O(b \log m)$ comparisons and $O(b)$ moves. Therefore, by Lemma 5 we can conclude that:

Lemma 6. *Under Assumption* 1, *b elements can be sorted stably, using* $O(1)$ *auxiliary space and another set of b distinct elements as placeholders, with* $O(b \log m)$ *comparisons and* $O(b)$ *moves.*

5.2. *The Fragmented Multi-Way Merging*

We know by hypothesis that the sequence $J'''P'''CB$ satisfies Property 1 and $b = |B| = \lceil |CB| / \log^3 m \rceil$. In Section 5.1 we showed how to sort b elements stably, using $O(1)$ auxiliary space, with $O(b \log m)$ comparisons and $O(b)$ moves, under Assumption 1 and using B as an internal buffer.

Now, we show how to sort CB using the technique in Section 5.1 and a multi-way stable merging technique requiring a very limited number of placeholders.

We want to solve the following problem:

Problem 4. We have

- $s \leq \log m / \log \log m$ sorted sequences $E_1, \ldots, E_s$ of $k \leq m/s$ elements each and
- a set $\mathscr{U}$ of $s(\lceil \log m \rceil)^2$ distinct elements.

Under Assumption 1, we want to sort the sk elements stably, using $O(1)$ auxiliary locations, with $O(sk \log m)$ comparisons and $O(sk)$ moves.

We name our solution to Problem 4 *fragmented multi-way merging*. Each sorted sequence is divided into $\gamma = k/\lceil \log m \rceil^2$ *fragments* of $\lceil \log m \rceil^2$ contiguous elements each (for simplicity, let us suppose $\lceil \log m \rceil^2$ divides k). Starting from the fragment with the largest element, we will denote the jth fragment of the sequence E_i with F_i^j.

The fragments of E_i are linked in a *bidirectional list* following the reverse sorted order of E_i. The fragment with the largest element of a sequence is the *head* of the list. For each list we need $2k/\lceil \log m \rceil^2$ words of $\lceil \log m \rceil$ bits to store the pointers; for that, we use $\mathbb{M}$ in the usual way.

One of the basic events in the process we are about to describe is the exchange of fragments (possibly belonging to two different sorted sequences). From now on we will assume that, when a fragment is moved, the pointers of its successor and its predecessor (if any) in its linked list are updated.

Let us denote the whole sequence of elements with P and with P_i the ith fragment of P from the left end, for $i = 1, \ldots, s$. The initial configuration is

$$P = E_1 E_2 \cdots E_{s-1} E_s U, \qquad \text{where} \quad E_i = F_i^{\gamma} F_i^{\gamma-1} \cdots F_i^2 F_i^1,$$

and U contains the set $\mathscr{U}$ in some order. First, we exchange F_1^1 with F_1^{γ}, F_2^1 with $F_1^{\gamma-1}$ and so forth until the s heads are the first s fragments of P ($P_1 = F_1^1$, $P_2 = F_2^1 \cdots$).

For $i = 1, \ldots, s$ the fragment P_i is associated with a small integer p_i of $O(\log \log m)$ bits containing the index (in P_i) of the first (from the right end) element of P_i not in $\mathscr{U}$. Two more indices *num* and *last* are maintained: *num* stores the current number of merged elements and *last* stores the address of the rightmost (in the whole P) fragment. All the integers p_i are stored in $\mathbb{M}$ while *num* and *last* are stored in *two normal locations of memory*. Initially all the small indices are set to $\lceil \log m \rceil^2$, *num* is set to 0 and *last* to $|P| - |U| - \lceil \log m \rceil^2 + 1$.

Then the merging phase begins. The following steps are repeated until $num = sk$:

1. The largest element among $P_1[p_1]$, $P_2[p_2]$, $\ldots$, $P_s[p_s]$ is found (for the stability, in case of equal elements the one in the fragment of the sorted sequence with the largest index is chosen). Let it be $P_i[p_i]$.

2. $P_i[p_i]$ is exchanged with $P[|P| - num]$, p_i is decreased by one and num is increased by one.
3. If P_i contains only elements of $\mathscr{U}$ (that is, $p_i = 0$), then let v be the address of the next fragment of E_i. P_i is exchanged with the fragment starting at v and then the fragment starting at v is exchanged with the one starting at $last$. Finally, we set $last = last - (\lceil \log m \rceil)^2$.

After the execution, $P = US$, where in U we have the elements of $\mathscr{U}$ in some order and in S we have the stably sorted sequence of the sk elements.

We prove that the wanted bounds hold. For any $1 \leq i \leq s$, step 1 requires $O(\log \log n)$ comparisons to decode the small integer p_i and one comparison between $P_i[p_i]$ and the current maximum. Since $s \leq \log m / \log \log m$, the total cost of each execution of step 1 is $O(\log m)$.

In step 2 we pay one exchange, one arithmetical increment and the cost of decreasing by one the selected p_i. This would be $O(\log \log m)$ moves *in the worst case* but only $O(1)$ moves *in the amortized sense* with an analysis similar to the one for a binary counter (see [2]).

Step 3 requires the access of a constant number of encoded pointers of $O(\log m)$ bits each and the exchange of a constant number of blocks of $\lceil \log m \rceil^2$ elements each (fragments). Hence the worst-case bounds are $O(\log m)$ for the comparisons and $O(\log^2 m)$ for the moves. However, step 3 is executed only when one of the head fragments $P_1, \ldots, P_s$, say P_i, is exhausted (i.e., it is full of buffer elements). Hence, the cost of the block exchanges of step 3 can be charged over the $\lceil \log m \rceil^2$ previous extractions from P_i that did not lead to the execution of step 3. Therefore $O(1)$ moves in amortized sense are performed in this step.

Since the total number of iterations is $O(sk)$, we have the wanted bounds and Problem 4 is solved.

Sorting CB, finally. With the fragmented multi-way merging and the technique of Section 5.1, we can finally sort the sequence CB when $b = |B| = \lceil |CB| / \log^3 m \rceil$.

1. C is logically divided into $t = \lceil |C| / b \rceil$ subsequences $C_1 C_2 \cdots C_{t-1} C_t$ of b elements each. Every C_i is sorted using the technique in Section 5.1 with B as internal buffer.
2. Since $b = \lceil |CB| / \log^3 m \rceil$, we have that $t = O(\log^3 m)$. Therefore, the sorted runs $C'_1 C'_2 \cdots C'_{t-1} C'_t$ can be merged executing a constant number of iterations of the multi-way mergesort using the fragmented multi-way merging with $s = \log m / \log \log m$ (we have plenty of distinct buffer elements to use with the fragmented multi-way merging since $|B| = \lceil |CB| / \log^3 m \rceil$).
3. B is sorted with the stable, in-place mergesort (using [17]) and C and B are merged in place and stably (using [17], again).

Therefore, by Lemma 6 and by the solution to Problem 4, we can conclude that:

Theorem 2. *Under Assumption* 1, *the subsequence CB of a sequence $J'''P'''CB$ satisfying Property* 1 *and $|B| = \lceil |CB| / \log^3 m \rceil$, can be sorted stably, using $O(1)$ auxiliary locations, performing $O(m \log m)$ comparisons and $O(m)$ moves in the worst case.*

6. Sorting with Few Distinct Elements

In this section we show how to sort the subsequence CB of a sequence $J'''P'''CB$ satisfying Property 1 and with $b = |B| < \lceil |CB|/\log^3 m \rceil$, that is, when the number d of distinct elements in CB is less than $\lceil |CB|/\log^3 m \rceil$. First, we solve a general problem in Section 6.1. Then, in Section 6.2, we show how the solution to the general problem can be used to sort CB.

6.1. *Sorting with Two Kinds of Internal Buffer*

We are interested in solving the following problem:

Problem 5. We are given:

- A set $\mathscr{D}$ of $d' < \lceil r/\log^2 m \rceil$ elements.
- Two sequences V and G of $t \leq m$ elements each and where V has $d'' \leq d'$ distinct elements.
- An $O(1)$-time Boolean function BELONGS_TO_$V(x)$ that, at any time, returns true if and only if x belongs to the set of elements originally contained in V.

We want to go from sequence VGD to sequence $V'GD'$ where V' contains the elements in V sorted stably and D, D' contain the elements in $\mathscr{D}$ in any order. Under Assumption 1, we have to use $O(1)$ auxiliary locations and perform $O(t \log m)$ comparisons and $O(t)$ moves.

The abstract problem can be seen as the problem of sorting a sequence V with few distinct elements having by our side (i) a constant time function helping to discern between the elements of V and the other ones and (ii) *two kinds of internal buffers*:

- The first buffer is small and the order of its elements is not important and can be lost after the process. Moreover, the number of elements in this buffer is greater than or equal to the number of distinct elements in V. That sequence would be D with the elements of set $\mathscr{D}$. When we use our solution for this abstract problem to sort the subsequence CB, the role of D obviously will be played by the subsequence of distinct element B.
- The second buffer is as large as V but the original order of its elements is important and has to be maintained after V is sorted. That large buffer would be G.

Our solution to Problem 5 has three phases.

6.1.1. *First Phase.* V is logically divided into $|V|/d'\lceil \log m \rceil^2$ contiguous blocks $V_1 V_2 \cdots$ of $d'\lceil \log m \rceil^2$ elements each. We want to sort any block V_i stably, using $O(1)$ auxiliary locations, $O(|V_i| \log m)$ comparisons and $O(|V_i|)$ moves. This can be accomplished in the same way that we sorted the sequence C in Section 5:

1. Each sub-block of d' contiguous elements of V_i is sorted using the d' elements of $\mathscr{D}$ as placeholders (Section 5.1).
2. The $\lceil \log m \rceil^2$ sorted sub-blocks of V_i are merged with a constant number of iterations of the multi-way mergesort using the fragmented multi-way merging

(Section 5.2). We have to distinguish two cases, though:

(a) If $|\mathscr{D}| = d' \geq \lceil \log m \rceil^3 / \log \log m$, we can apply the solution to Problem 4 as presented in Section 5.2.

(b) If, on the other hand, $|\mathscr{D}| = d' < \lceil \log m \rceil^3 / \log \log m$ we may not have a sufficient number of distinct elements for the set $\mathscr{U}$ in Problem 4. However, if $d' < \frac{\lceil \log m \rceil^3}{\log \log m}$ then
$|V_i| = d' \lceil \log m \rceil^2 = \text{polylog}(m)$.
Hence, we can use the same fragmented multi-way merging technique in Section 5.2 but with fragments of size $O(\log \log m)$ instead of $O(\log^2 m)$. That reduces the size of the set $\mathscr{U}$ from $O(s \log^2 m)$ to $O(s \log \log m)$. If we choose $s = \log m / (\log \log m)^2$, the number of iterations of the s-way mergesort needed to sort the whole block V_i is still a constant but the size of $\mathscr{U}$ is $O(\log m / \log \log m)$. Therefore, the elements in $\mathscr{U}$ do not have to be distinct anymore because we can maintain in a single word of (actual) auxiliary memory the whole permutation to bring them back to their original order when the fragmented s-way merging process is done.

6.1.2. *Second Phase*. After the first phase, each block V_i of V is sorted and divided into at most $d'' \leq d'$ runs of equal elements. Since $|V_i| = d' \lceil \log m \rceil^2$, the total number t_r of runs in V is less than or equal to $t / \lceil \log m \rceil^2$. For any run, let the first element be the *head* and the rest of the run be the *tail*. The second phase has four main steps:

1. Each block V_i is divided into two sub-blocks H_i and V_i'. H_i contains the heads of all the runs of V_i and V_i' contains all the tails. Both H_i and V_i' are in sorted order. This subdivision can be accomplished in a linear number of moves with at most d'' applications of the well-known in-place block-exchanging technique (recalled in Section 4.1).

   Let i_r be the number of runs of V_i. Let $h_1, \ldots, h_{i_r}$ be the heads we have to collect, indexed from the leftmost to the rightmost in V_i.

   Let $U_1, \ldots, U_{i_r}$ be the subsequences of V_i that separate $h_1, \ldots, h_{i_r}$, that is

   $$V_i = h_1 U_1 h_2 U_2, \ldots, U_{i_r - 1} h_{i_r} U_{i_r}$$

   (some of them can be void). We collect $h_1, \ldots, h_{i_r}$ in a growing subsequence H_i starting from the position of h_1. During the process, H_i slides toward the right end of V_i. The process scans V_i from left to right and therefore the positions of $U_1, \ldots, U_{i_r}$ and $h_1, \ldots, h_{i_r}$ are obtained "on the fly," during the scan.

   Let $H_i = h_1$ and $j = 1$. The following steps are repeated until $j > i_r$:

   (a) If $|H_i| \leq |U_j|$, do a block exchange between the two adjacent blocks H_i and U_j.

   (b) Otherwise, let $H_i = H_i' H_i''$ with $|H_i'| = |U_j|$. Exchange H_i' with U_j (obvious exchange of two non-adjacent but equal sized blocks). After that $H_i = H_i'' H_i'$.

   (c) In both cases, now H_i is adjacent to h_{j+1}; let $H_i = H_i h_{j+1}$ and increase j by one.

   Since the elements we are extracting (the heads of the runs of a single block V_i) are distinct, we do not care about their original order during the process. We simply sort them when they are finally collected at the right end of V_i. On the

other hand, the order of the other elements of the runs of V_i is maintained in the process.

2. Some information about runs and blocks is collected and stored in $\mathbb{M}$.
   - An array I_H with $|V|/d'\lceil\log m\rceil^2$ entries of two words each is stored in $\mathbb{M}$. For any i, the first word of $I_H[i]$ contains $|H_i|$ and the second word contains the index of the first run of V_i (the index is between 1 and t_r, from the leftmost run in V to the rightmost).
   - An array I_R with t_r entries of four words each is stored in $\mathbb{M}$. For any i, the first word of $I_R[i]$ is initially set to i, the second contains the address of the head of the ith (in V) run, the third contains the starting address of the tail of the ith run and the fourth contains the size of the ith run.
   - Finally, an array $I_{R^{-1}}$ with t_r entries of two words each is stored in $\mathbb{M}$. For any i, the first word of $I_{R^{-1}}[i]$ is initially set to i and the second word of $I_{R^{-1}}[i]$ is initially set to 1.

   All this information can be obtained within our target bounds simply by scanning V. In general, for any array I of multi-word entries, we will denote the pth word of the ith entry with $I[i][p]$.
3. $I_{R^{-1}}$ is sorted stably by head, that is, at any time of the sorting process, the sorting key for the two-word value in the ith entry of $I_{R^{-1}}$ is

   $$V[I_R[I_{R^{-1}}[i][1]][2]].$$

   The sorting algorithm used is mergesort with a linear-time, in-place stable merging (e.g., the one described in [17]). During the execution of the algorithm, *every time the two-word value in the* ith *entry of* $I_{R^{-1}}$ *is moved to the* jth *entry, the corresponding entry in* I_R *is updated, that is,* $I_R[I_{R^{-1}}[j][1]][1]]$ *is set to* j.

   We remark that *only the entries of the encoded array* $I_{R^{-1}}$ *are moved* (where any abstract move of an encoded value causes $O(\log m)$ actual moves of some elements contained in zones Q' and Q'' defined in Section 3). In this process, the elements in V are not moved.
4. For $i = 2$ to t_r, let $I_{R^{-1}}[i][2]$ be $I_{R^{-1}}[i-1][2] + I_R[I_{R^{-1}}[i][1]][4]$ (that is, if we had the elements in V sorted stably into another sequence V', $I_{R^{-1}}[i][2]$ would be the starting address in V' of the ith run in the stable sorted order).

6.1.3. *Third Phase.* After the second phase we are able to evaluate the function α_V: $\{1,\ldots,t\} \to \{1,\ldots,t\}$ such that $\alpha_V(j)$ is the rank of the element $V[j]$ in the sequence V, performing $O(\log m)$ comparisons.

1. Let V_i be the block of $V[j]$. We know where H_i starts and ends, in fact

   $$H_i = V[s_i \ldots s_i + I_H[i][1] - 1] \qquad \text{where} \quad s_i = (i-1)d'(\lceil\log m\rceil)^2 + 1.$$

   Therefore, we can perform a binary search for $V[j]$ in H_i and find the index p_j in V_i of the run to which $V[j]$ belongs.
2. The index p'_j in V of the run of $V[j]$ is $I_H[i][2] + p_j - 1$.
3. Using the array I_R, we can find the position k_j of $V[j]$ in its run. If $j = I_R[p'_j][2]$ then $V[j]$ is the head of its run. Otherwise, $V[j]$ belongs to the tail of its run.

Let us define k_j in the following way:

$$k_j = \begin{cases} 1 & \text{if} \quad j = I_R[p'_j][2], \\ j - I_R[p'_j][3] + 2 & \text{otherwise.} \end{cases}$$

4. Finally, we have that $\alpha_V(j) = I_{R^{-1}}[I_R[p'_j][1]][2] + k_j - 1$.

Using this algorithm and the given function BELONGS_TO_$V(x)$ to discern between the elements originally contained in V and the ones originally in G, it is possible to sort the elements in V efficiently, using G as the internal buffer while preserving the original order of its elements.

The idea. Before the formal description of this last phase is given, a short outline is needed. The algorithm has two nested iterations:

- The outer iteration scans the elements of V following the sorted order (we know the order of the runs from the previous phase, therefore the elements can be scanned in sorted order easily).

  During the scan, three kinds of elements can be found: (i) heads of runs, (ii) elements belonging to the tails of their runs and (iii) buffer elements from G. (As we will see, the inner iteration is responsible for the presence of these elements.)
  - If a buffer element is found (recognized using the given function BELONGS_TO_$V(x)$), there is nothing to do: the element of V previously stored in this position has already reached its final destination.
  - If a head is found, nothing can be done since the heads are the cornerstones of the algorithm used to find the rank of an element in V. As we will see, their treatment is delayed until the very end of the algorithm.
  - Finally, if an element x of a tail is found, the inner iteration starts.
- The purpose of the inner iteration is to scan the cycle (of the permutation that disposes the elements of V in sorted order) to which the element belongs. During the scan of the cycle of x two kinds of elements can be found: (i) heads of runs and (ii) tail elements. (Obviously, the first found is x.) Again, the heads are left in their position. On the other hand, any tail element y is ranked (with α_V); let its rank in V be r_y, and it is exchanged with the element in G corresponding to its rank, that is, $b_y = G[r_y]$. Then there can be two cases:
  - If $V[r_y]$ is a head, it cannot be moved and b_y is left in the position in V previously occupied by y and is treated in a special way.
  - If $V[r_y]$ is a tail element, we immediately exchange $V[r_y]$ with b_y, recovering the correct position for b_y. Therefore, the next element of the cycle is in the position previously occupied by y.

After the two nested iterations, a final simple iteration performs t_r exchanges that bring the heads to their final positions.

The Algorithm. Now we can give a precise description of the algorithm for the third phase of our solution to Problem 5.

The function *is_head*(x) used in the algorithm returns true if $V[x]$ is the head of its run. It can be calculated in the very same way as the rank of an element in V (with the

exclusion of the fourth step):

```
 1: FOR i = 1 to t_r and j = 2 to I_R[I_{R^-1}[i][1]][4] DO
 2:    start ← I_R[I_{R^-1}[i][1]][3] + j − 1
 3:    BELONGS_TO_V(V[start]) THEN
 4:       next ← α_V(start)
 5:       WHILE next ≠ start DO
 6:          Exchange V[start] and G[next]
 7:          IFis_head(next) THEN
 8:             next ← α_V(next)
 9:          ELSE
10:             next_tmp ← next
11:             next ← α_V(next)
12:             Exchange V[start] and V[next_tmp]
13: FOR i = 1 to t_r DO
14:    head ← I_R[I_{R^-1}[i][1]][2]
15:    Exchange V[head] and G[I_{R^-1}[i][2]]
16: Exchange G and V
```

6.2. *Sorting CB*

With the solution to the abstract Problem 5, we can finally sort the sequence CB when $b = |B| < \lceil |CB|/\log^3 m \rceil$:

1. C is partitioned into three subsequences $C'UC''$, where U contains all the elements equal to the element c_m of rank $\lceil |C|/2 \rceil$ in C, and C', C'' contain all the elements of C, respectively, less than and greater than c_m. This partition can be easily obtained using the stable, in-place selection and the stable, in-place partitioning in [7] and [8].
2. To sort C' and C'' we can apply the solution to Problem 5:
   (a) We set $V = C'$, $G = (UC'')[1 \cdots |C'|]$, $D = B$, BELONGS_TO_$V(x) = (x < c_m)$ and sort C'.
   (b) We set $V = C''$, $G = (C'U)[1 \cdots |C''|]$, $D = B$, BELONGS_TO_$V(x) = (c_m < x)$ and sort C''.
   (Obviously there can be extreme situations in which C' or C'' are void.)
3. B is finally sorted with the normal mergesort using a linear-time, in-place, stable merging (e.g., the one described in [17]) and the two sequences are merged (once again with the algorithm given in [17]).

Therefore, we can conclude that:

Theorem 3. *Under Assumption* 1, *the subsequence CB of a sequence $J'''P'''CB$ satisfying Property* 1 *and $|B| < \lceil |CB|/\log^3 m \rceil$ can be sorted stably, using $O(1)$ locations of auxiliary memory, performing $O(m \log m)$ comparisons and $O(m)$ moves in the worst case.*

7. Conclusion

By Theorems 1–3 we can conclude that Problem 1 is solved and state the main result of this paper.

Theorem 4. *Any sequence of n elements can be sorted stably, using* $O(1)$ *auxiliary locations of memory, performing* $O(n \log n)$ *comparisons and* $O(n)$ *moves in the worst case.*

This settles a long-standing open question explicitly stated by Munro and Raman in [13]. Before the introduction of this algorithm, the best-known solution for stable, in-place sorting with $O(n)$ moves was the one presented in [14], performing $O(n^{1+\varepsilon})$ comparisons in the worst case.

References

[1] H. Bing-Chao and D. E. Knuth. A one-way, stackless quicksort algorithm. *BIT*, 26(1):127–130, 1986.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2001.

[3] B. Ďurian. Quicksort without a stack. In B. Rovan J. Gruska and J. Wiedermann, editors, *Proceedings of the* 12*th Symposium on Mathematical Foundations of Computer Science*, volume 233 of LNCS, pages 283–289. Springer-Verlag, Berlin, 1986.

[4] G. Franceschini and V. Geffert. An in-place sorting with $O(n \log n)$ comparisons and $O(n)$ moves. In *Proceedings of the* 44*th Annual IEEE Symposium on Foundations of Computer Science* (*FOCS*), pages 242–250, 2003.

[5] C. A. R. Hoare. Quicksort. *Comput. J.*, 5(1):10–16, April 1962.

[6] A. Itai, A. G. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In *ICALP* '81, volume 115 of LNCS, pages 417–31. Springer-Verlag, Berlin, 1981.

[7] J. Katajainen and T. Pasanen. Sorting multisets stably in minimum space. In O. Nurmi and E. Ukkonen, editors, *SWAT* '92, volume 621 of *LNCS*, pages 410–421. Springer-Verlag, Berlin, 1992.

[8] J. Katajainen and T. Pasanen. Stable minimum space partitioning in linear time. *BIT*, 32(4):580–585, 1992.

[9] J. Katajainen and T. Pasanen. In-place sorting with fewer moves. *Inform. Process. Lett.*, 70:31–37, 1999.

[10] D. E. Knuth. *The Art of Computer Programming*, *Vol.* 3: *Sorting and Searching*. Addison-Wesley, Reading, MA, 1973.

[11] M. A. Kronrod. Optimal ordering algorithm without operational field. *Soviet Math. Dokl.*, 10:744–746, 1969.

[12] J. I. Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *J. Comput. System Sci.*, 33:66–74, 1986.

[13] J. I. Munro and V. Raman. Sorting with minimum data movement. *J. Algorithms*, 13:374–93, 1992.

[14] J. I. Munro and V. Raman. Fast stable in-place sorting with $O(n)$ data moves. *Algorithmica*, 16:151–60, 1996.

[15] J. I. Munro and V. Raman. Selection from read-only memory and sorting with minimum data movement. *Theoret. Comput. Sci.*, 165:311–23, 1996.

[16] L. T. Pardo. Stable sorting and merging with optimal space and time bounds. *SIAM J. Comput.*, 6(2):351–372, June 1977.

[17] J. Salowe and W. Steiger. Simplified stable merging tasks. *J. Algorithms*, 8(4):557–571, December 1987.

[18] L. M. Wegner. A generalized, one-way, stackless quicksort. *BIT*, 27(1):44–48, 1987.

[19] J. W. J. Williams. Heapsort (Algorithm 232). *Comm. ACM*, 7:347–48, 1964.

Received January 19, 2006. *Online publication March* 1, 2007.